1 INTRODUCTION

Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a dénouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations.

— MR. Cox ***1893***, *Cinderella: Three hundred and forty-five variants [**?**]*

THE first web browser, Nexus, made its appearance in 1990 [?]. At its inception, web browsing consisted solely of retrieving and displaying small, static text pages. With Nexus, users could access for the first time interlinked hypertext documents, so-called HTML pages. However, the escalating computing power of devices, the proliferation of the internet, the valuation of internet-based companies and the demand for more engaging user experiences gave rise the concept of executing code in conjunction with web pages. In 1995, the Netscape browser revolutionized this concept by introducing JavaScript [?], a programming language that allowed code execution on the client-side. Interactive web content immediately highlighted benefits: unlike classical native software, web applications do not require installation, are always up-to-date, and are accessible from any device with a web browser. Significantly, since the advent of Netscape, all browsers offer JavaScript support. In the present day, the majority of web pages incorporate not only HTML but also JavaScript code, which is executed on client computers. Over the past several decades, web browsers have transformed into JavaScript virtual machines. They have evolved into intricate systems capable of running comprehensive applications, such as video and audio players, animation creators, and PDF document renderers.

Despite being the main scripting language in modern web browsers, JavaScript possesses inherent limitations due to its unique language characteristics [?]. Each JavaScript engine requires the parsing and recompiling of the JavaScript code, thereby causing substantial overhead. In practice, the process of parsing and compiling JavaScript code constitutes the majority of website load times [?]. Additionally, JavaScript presents security issues, including the lack of memory

[0] Compilation probe time 2023/11/09 15:07:44

isolation, which potentially enables information extraction from other processes [? ?]. Numerous attempts have been made to port other languages, offering different guarantees, to the browser execution as alternatives to JavaScript. For instance, Java applets emerged on web pages in the late 90s, enabling the execution of Java bytecode on the client side [?]. Likewise, Microsoft attempted twice with ActiveX in 1996 [?], and Silverlight in 2007 [?]. However, these attempts either failed to persist or experienced low adoption, primarily due to security issues and the absence of consensus among the community of browser vendors.

Importantly, in 2014, Alon Zakai and colleagues proposed the Emscripten tool [?]. Emscripten employs a strict subset of JavaScript, asm.js, to facilitate the compilation of low-level code such as C to JavaScript. Asm.js was included as an LLVM backend [?]. This strategy offered the advantages of the ahead-of-time optimizations from LLVM, resulting in performance gains on browser clients [?] when compared to standard JavaScript code. Asm.js outperformed JavaScript because it restricted language features to those that could be optimized in the LLVM pipeline. Moreover, it eliminated most of the language's dynamic characteristics, limiting it to numerical types, top-level functions, and one large array in memory accessed directly as raw data. Asm.js proved that client-side code could be enhanced with the appropriate language design and standardization. In response to persistent JavaScript-related issues, the formalization and creation of a formal specification following asm.js laid the groundwork for the emergence of WebAssembly. In 2015, the Web Consortium (W3C) standardized WebAssembly as a bytecode for the web environment. As a result, WebAssembly bytecode became the fourth official language for the web.

The first distinction from earlier attempts to port non-JavaScript languages to the web lies in WebAssembly's initial design. Unlike its predecessors, WebAssembly was crafted to supplement JavaScript in the browser as a platform-agnostic, low-level bytecode, rather than to completely replace it. Its primary goal was to replace computing-intensive JavaScript code in contemporary web applications. Additionally, WebAssembly is the inaugural major language that utilized formal specification and verification right from the design inception [? ? ?].

Importantly, WebAssembly provides a platform for compiling several legacy code applications, like those written in C/C++. For example, LLVM includes WebAssembly as a backend since release 7.1.0 published in May 2019[1]. Therefore, the emergence of WebAssembly, a swift, low-level, portable bytecode for browsers, has the potential to transform web software as we know it. It paves the way for web applications to undertake roles traditionally reserved for native desktop applications. For example, applications such as AutoCAD and Adobe Photoshop have been ported to WebAssembly[2].

[1] https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0

[2] https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R6

The WebAssembly specification embodies several language design principles that pave the way for its extension beyond the web ecosystem. For instance, the architecture of WebAssembly guarantees self-containment. Inherently, WebAssembly binaries are prohibited from accessing memory beyond their own designated space, thereby amplifying security via binary isolation from the remainder of the system. Consequently, research has highlighted the benefits of integrating WebAssembly as an intermediate layer in contemporary cloud platforms [?]. In particular, the employment of WebAssembly binaries expedites startup times and optimizes memory consumption, outperforming virtualization and containerization [?]. Furthermore, in comparison to virtual machines and containers, they are more compact in size, highlighting their efficient deployment, especially when network transportation is a consideration. The methodology for standalone WebAssembly execution was formalized in 2019. The Bytecode Alliance unveiled the WebAssembly System Interface (WASI) [? ?]. WASI standardizes the execution of WebAssembly via a POSIX system interface protocol, thereby facilitating the direct execution of WebAssembly within the operating system. This standardization enables WebAssembly to function outside the confines of the browser, extending its use to cloud environments and IoT devices, for example.

The extensive applicability and rapid adoption of WebAssembly have prompted requests for additional features. However, these demands do not always align with the initial specifications. For a proposal to reach standard status, it must satisfy particular criteria. A new proposal needs a formal specification and a minimum of two independent implementations, e.g., two different WebAssembly engines. This approach allows for swift incorporation of new formalizations and features via the so-called "evergreen method", while maintaining the original WebAssembly specification intact. Consequently, since the inception of WebAssembly, numerous extensions have been proposed for standardization. For instance, the SIMD proposal enables the execution of vectorized instructions in WebAssembly. After approval, new extensions remain optional, ensuring that the core WebAssembly version remains 1.0. The ongoing development of WebAssembly provides avenues for research and development. However, it also gives rise to security concerns within the ecosystem, as new threats emerge.

1.1 The risks of WebAssembly monoculture

Web browsers and JavaScript have evolved significantly in the past three decades, leading to numerous implementations. Yet, only Firefox, Chrome, Safari, and Edge are commonly used on devices. This situation reflects a software monoculture problem wherein a single flaw could impact multiple devices at the

9lw

same time. The concept of monoculture is borrowed from biology and symbolizes an ecosystem at risk of extinction due to shared vulnerabilities and lack of diversity. Currently, web pages including WebAssembly binaries are centrally served from main datacenters. Thus, this monoculture issue is also applicable to the WebAssembly code served to web browsers. Therefore, sharing Wasm code through web browsers could also share its potential vulnerabilities. The software monoculture problem is exacerbated when considering the edge-cloud computing platforms and their adoption of WebAssembly to provide services. Specifically, in addition to browser clients, thousands of edge devices running WebAssembly as backend services could be affected by shared vulnerabilities. This scenario suggests that if one node in an edge network is vulnerable, all the others would be vulnerable in the exact same way since the same binary is replicated on each node. The same attacker payload could compromise all edge nodes simultaneously, meaning that a single distributed Wasm binary could trigger a worldwide attack.

Generally, a software monoculture facilitates the spread of vulnerabilities. Despite the praise for WebAssembly's security, particularly its design that prohibits programs from accessing data beyond their own memory, it is not immune to vulnerabilities. For instance, vulnerabilities and attacks within the memory of WebAssembly itself are feasible[**?**]. Intriguingly, innate vulnerabilities can exist in WebAssembly binaries due to flaws in the source code. For instance, the lack of stack-smashing protections could result in unnoticed overflows and crashes during WebAssembly executions[**?**]. Notably, there are significant risks associated with side-channel attacks on WebAssembly. In standalone deployments, Genkin et al. demonstrated the possibility of data extraction via cache-timing side channels in WebAssembly [**?**]. In a similar vein, Maisuradze and Rossow exhibited speculative execution attacks on WebAssembly binaries [**?**].

On the other hand, malware can also be deployed through WebAssembly binaries. Defenders often assume the existence of a single malware binary variant, while in reality, numerous variants of the same malware might exist. For example, as WebAssembly's popularity increases, it is more frequently employed in computation-intensive tasks within browsers, including gaming and image processing. This enhanced capability for efficient computation simultaneously makes WebAssembly a desirable target for misuse by cybercriminals, especially for cryptojacking [**?**]. The difficulty in detecting and eliminating cryptojacking allows it to operate persistently on a victim's computer, continually using resources and generating income for the attacker [**?**]. Additionally, the complexity of WebAssembly code enhances its suitability for obfuscating malicious code, including JavaScript malicious payloads [**?**]. A variety of techniques, including static analysis, dynamic analysis, and even advanced machine learning methods, are employed to detect WebAssembly crytomalware [**? ? ? ? ? ?**]. However, most of these studies do not consider the presence of obfuscation tools, which poses a significant, largely unexplored threat to the accuracy of malware detection in WebAssembly. Rokicki et al. have also highlighted the potential risk of port contention side-channel attacks using WebAssembly malware in browsers [?]. In such cases, mitigations often involve hardware and operating-level changes, which are not always feasible.

1.2 Problems statements

To sum up, software monoculture amplifies the dissemination of vulnerabilities. The effect of exploiting a single vulnerability in WebAssembly could prove catastrophic, given all devices running WebAssembly binaries could be affected. Conversely, WebAssembly malware pose a severe threat. Present defenses may not adequately protect against them, as they have not been designed to manage situations outside of software monoculture scenarios, such as obfuscation. Besides, mitigations might require hardware and operating-level changes, which are not always feasible. Last but not least is the lack of Software Diversity research in the context of WebAssembly. In this dissertation, we tackle the subsequent three problems:

P1 Side-channels and persisting vulnerabilities: The WebAssembly ecosystem and binaries are susceptible to attacks, especially those from side-channel threats.

P2 WebAssembly malware: WebAssembly malware presents a substantial threat. The phenomenon of software monoculture makes to assume that malware is typically considered only within the context of the original program.

P3 Inexisting diversification tools for WebAssembly: Software Diversity research in the context of WebAssembly is unexplored.

1.3 Software Diversification

This dissertation introduces tools, strategies, and methodologies designed to address the previously enunciated problem statements via Software Diversification. Software Diversification is a security-focused process that involves identifying, developing, and deploying program variants of a given original program [?]. Pioneers in this field, Cohen et al. [?] and Forrest et al. [?], proposed enhancing software diversity through code transformations. Their proposal suggested creating program variants while maintaining their functionalities to mitigate potential vulnerabilities.

Software diversification has been proven to effectively remove vulnerabilities, as evidenced by previous studies. For instance, Eichin et al. [?], in their seminal 1989 work, highlighted the practical benefits of diversification. They demonstrated that diversification constrained the exploitation of the Morris Worm to a handful of machines. Additionally, software diversification could bolster WebAssembly analysis tools by integrating diversified program variants, thereby making it harder for attackers to exploit vulnerabilities, addressing **P1**. On the other hand, by proactively creating variants for security purposes, a broad range of real-world conditions can be emulated, enhancing the precision of WebAssembly analysis tools, including WebAssembly malware detectors. Furthermore, it could increase the accuracy of WebAssembly malware detectors and WebAssembly analysis tools in general, addressing **P2**. Besides, the diversification process could completely eradicate malware from one variant to the next by removing malicious code from the original program, addressing **P2**. However, despite the extensive research, the implementation of software diversification in WebAssembly is still largely unexplored. In light of this, we offer the following contributions within the context of software diversification (**P3**), which are not necessarily mutually exclusive.

C1 Defensive Diversification: In order to address **P1**, we assess how generated WebAssembly program variants could be used for defensive purposes. We provide empirical insights about the practical usage of the generated variants in preventing attacks.

C2 Offensive Diversification: In order to address **P2**, we evaluate the potential for using generated WebAssembly program variants for offensive purposes. Our research includes experiments where we test the resilience of WebAssembly analysis tools against these generated variants. Furthermore, we offer insights into which types of program variants practitioners should prioritize to improve WebAssembly analysis tools.

C3 Experimental contribution: For each proposed technique we provide an artifact implementation and conduct experiments to assess its capabilities. The artifacts are publicly available. The protocols and results of assessing the artifacts provide guidance for future research on **P1**, **P2** and **P3**.

C4 Theoretical contribution: We propose a theoretical foundation in order to generate and improve Software Diversification for WebAssembly, addressing **P3**. We provide a formal definition of WebAssembly program variants and their diversity. We also provide a formal definition of WebAssembly program diversity generation.

C5 Diversity generation: To address and asses **P3**, we generate WebAssembly program variants. The variants are functionally equivalent to the original program, yet behaviorally diverse.

Contribution	Research papers			
	I [?]	II [?]	III [?]	IV [?]
C1 Defensive diversification	✓	✓	✓	
C2 Offensive diversification				✓
C3 Experimental contribution	✓	✓	✓	✓
C4 Theoretical contribution	✓		✓	
C5 Diversity generation	✓	✓	✓	✓

Table 1.1: *Mapping between contributions and research papers .*

1.4 Summary of research papers

This compilation thesis comprises the following research papers. In Table 1.1 we map the contributions to our research papers.

I: **CROW: Code randomization for WebAssembly bytecode.**
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages
`https://doi.org/10.14722/madweb.2021.23004`

Summary: In this paper, we introduce the first entirely automated workflow for diversifying WebAssembly binaries. We present CROW, an open-source tool that implements software diversification through enumerative synthesis. We assess the capabilities of CROW and examine its application on real-world, security-sensitive programs. In general, CROW can create many statically diverse variants. Furthermore, we illustrate that the generated variants exhibit different behaviors at runtime.

II: **Multivariant execution at the Edge.**
Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Moving Target Defense (MTD 2022), 12 pages
`https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

Summary: In this paper, we synthesize functionally equivalent variants of deployed edge services. Service variants are encapsulated into a single multivariant WebAssembly binary. A random variant is selected and executed each time a function is invoked. Execution of multivariant binaries occurs on the global edge platform provided by Fastly, as part of a research collaboration. We demonstrate that multivariant binaries present a diverse range of execution traces throughout the entire edge platform, distributed worldwide, effectively creating a moving target defense.

III: **Wasm-mutate: Fast and efficient software diversification for WebAssembly.**
Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
Under review, 17 pages
`https://arxiv.org/pdf/2309.07638.pdf`

Summary: This paper introduces WASM-MUTATE, a compiler-agnostic WebAssembly diversification engine. The engine is designed to swiftly generate functionally equivalent yet behaviorally diverse WebAssembly variants by randomly traversing e-graphs. We show that WASM-MUTATE can generate tens of thousands of unique WebAssembly variants in minutes. Importantly, WASM-MUTATE can safeguard WebAssembly binaries from timing side-channel attacks, such as Spectre.

IV: **WebAssembly Diversification for Malware evasion.**
Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023, 17 pages

Summary: WebAssembly, while enhancing rich applications in browsers, also proves efficient in developing cryptojacking malware. Protective measures against cryptomalware have not factored in the potential use of evasion techniques by attackers. This paper delves into the potential of automatic binary diversification in aiming WebAssembly cryptojacking detectors' evasion. We provide proof that our diversification tools can generate variants of WebAssembly cryptojacking that successfully evade VirusTotal and MINOS. We further demonstrate that these generated variants introduce minimal performance overhead, thus verifying binary diversification as an effective evasion technique.

■ Thesis layout

This dissertation comprises two parts as a compilation thesis. Part one summarises the research papers included within, which is partially rooted in the author's licentiate thesis [**?**]. Chapter 2 offers a background on WebAssembly and the latest advancements in Software Diversification. Chapter 3 delves into our technical contributions. Chapter 4 exhibits two use cases applying our technical contributions. Chapter 5 concludes the thesis and outlines future research directions. The second part of this thesis incorporates all the papers discussed in part one.